Report finale PPS - Among Sus

Oleg Konchenkov
oleg.konchenkov@studio.unibo.it

Elia Pasqualini
elia.pasqualini@studio.unibo.it

Giovanni Poggi
giovanni.poggi2@studio.unibo.it

23 marzo 2021

Indice

Elenco delle figure

Introduzione

Il progetto si pone come obiettivo la realizzazione del videogioco Among Us, piattaforma multiplayer che supporta da 4 fino a 10 giocatori. Durante ogni partita, da 1 a 3 giocatori vengono selezionati casualmente e contrassegnati come *Impostor*, mentre gli altri saranno *Crewmate*. Gli *Impostor* avranno la capacità di sabotare i sistemi della mappa, attraversare le "botole" presenti in alcuni punti della mappa per passare da una stanza all'altra più rapidamente, identificare qualsiasi altro *Impostor* ed il loro scopo sarà quello di uccidere i *Crewmate*.
L'obiettivo dei *Crewmate* è scoprire gli *Impostor* ed eliminarli prima di essere assassinati o sabotati. Quando un giocatore muore, diventa un fantasma, il cui obiettivo è aiutare i membri rimanenti della propria squadra.
Se un giocatore trova un cadavere, può segnalarlo, il che porterà ad una riunione di gruppo in cui i giocatori discuteranno su chi pensano sia un *Impostor*, sulla base delle prove che hanno visto vicino l'omicidio. Se si raggiunge la maggioranza dei voti, la persona prescelta viene espulsa dalla mappa e verrà rivelato se era un *Impostor* o meno. I giocatori possono chiedere in qualsiasi momento, ma solo una volta a partita, un *"Emergency Meeting"* dirigendosi verso l'inizio della mappa ed attivando il bottone apposito.

Capitolo 1

Processo di sviluppo

È stato adottato un metodo di sviluppo agile, simile a **Scrum**, che consente di dividere un progetto per fasi, denominate *Sprint*, con ogni fase focalizzata su nuove funzionalità. I clienti, simulati dai membri del team, possono vedere il proseguimento del lavoro per verificare la soddisfazione, quindi apportare e richiedere modifiche in tempi più brevi. Ciò può aumentare l'efficienza del gruppo di lavoro, aumentando così la produttività.
La prima fase del processo di sviluppo è stata caratterizzata dalla modellazione dell'architettura generale del progetto. Sono stati definiti i requisiti ed in base a questi mediante diagrammi UML è stata effettuata una prima fase di progettazione. Come ultima cosa è stato creato un **Product Backlog** seguito dalla compilazione di un primo **Sprint Planning**.

1.1 Meeting

I meeting tra i membri del team sono avvenuti regolarmente mediante Microsoft Teams e la piattaforma Discord.
I componenti del team hanno partecipato attivamente ai meeting svolgendo tutti il ruolo di Sviluppatori mentre Elia Pasqualini ha ricoperto anche il ruolo di **Product Owner**.
All'inizio di ogni settimana sono stati effettuati in modo affine sia la **Sprint Review** che lo **Sprint Planning**. In questi incontri si è controllato il lavoro svolto nella settimana precedente e sono stati determinati obiettivi e compiti da svolgere per ciascun membro. Inoltre, ogni componente è stato aggiornato sullo stato di avanzamento dei *task* individuati e si è aggiornato di conseguenza il *Product Backlog*. Gli incontri svoltisi durante la settimana, sono serviti per aggiornare i membri del team sui progressi svolti e per far emergere eventuali problematiche riscontrate allo scopo di studiare insieme eventuali soluzioni, similmente a come avverrebbe nel tipico **Daily Scrum** a cui si è fatto riferimento.

1.2 Divisione dei task

Come prima cosa durante ciascun sprint settimanale si è fatto utilizzo del *Product Backlog* per assegnare a ciascun componente del team una serie di *task* da svolgere nei 7/10 giorni successivi, in modo da definire gli obiettivi che ogni componente del team è tenuto a portare a termine. Un *task* viene rappresentato da uno o più requisiti tra quelli individuati nel primo periodo del processo di sviluppo.

1.3 Revisione dei task

Durante la **Sprint Review**, dopo la conclusione di ciascuno sprint, i membri del team hanno informato gli altri componenti dei progressi nello svolgimento dei *task* a lui assegnati, notificando le varie difficoltà incontrate durante lo svolgimento di questi ultimi. A seguito del risultato della review, è stato aggiornato e revisionato il *Product Backlog* valutando eventuali ritardi sulla tabella di marcia e le relative contromisure da adottare. Solitamente, dopo il completamento dell'attività, prima della conclusione effettiva, viene eseguita un'operazione di revisione del codice, in cui il codice generato dal responsabile del *task* viene mostrato a tutti i membri del team, lo scopo è quello di determinare possibili fattori di refactoring per migliorare la qualità del codice prodotto, riducendo così il potenziale debito tecnico.

1.4 Scelta degli strumenti

Per supportare i processi di sviluppo e progettazione sono stati utilizzati diversi strumenti. Lo scopo dell'utilizzo di questi ultimi è quello di fornire agli sviluppatori servizi durante tutto il processo di sviluppo automatizzandoli, in modo da migliorarne l'efficienza e rendere il team più concentrato sulla soluzione delle esigenze del progetto stesso.

- **SBT**: Come strumento di **Build Automation**;
- **Scalatest**: Utilizzato per la scrittura e l'esecuzione dei Test Automatizzati;
- **GitHub Actions**: Come strumento di **Continuous Integration**, in modo da testare progetti software ospitati sul repository GitHub;
- **GitHub**: Come servizio di **Repository** del codice sorgente e file utilizzati durante il processo di sviluppo come, ad esempio, il *Product Backlog*;
- **FindBugs** e **Scalastyle** sono stati utilizzati a supporto del controllo di qualità automatizzato del codice;
- **Trello**: Per avere una panoramica dell'avanzamento complessivo del progetto e delle funzioni specifiche da eseguire o completare. Sono state create

etichette personalizzate da associare alle funzioni da implementare per organizzare al meglio il lavoro di ogni membro del team. Inoltre sono state create colonne nelle quali gli *item* da eseguire sono raggruppati in base al loro stato di sviluppo. Alcune di esse sono:

- **To-Do**: Contiene le schede relative alle funzioni ancora da sviluppare;
- **Progress**: Contiene le schede relative alle funzioni che si stanno realizzando in quel preciso istante;
- **Done**: Contiene le schede relative alle funzioni già sviluppate in precedenza;
- **Bug**: Contiene le schede relative alle funzioni che presentano errori da correggere;
- **Sprint**: Contiene il riepilogo di ciascuna funzionalità implementata in ogni Sprint.

Qui di seguito si può vedere il Link alla Bacheca *Trello*:
`https://trello.com/b/3stYMWBd/pps-among-sus-konchenkov-pasqualini-poggi`

Capitolo 2

Requisiti

Questo capitolo si propone di descrivere in dettaglio tutti i requisiti del software implementato. Dall'inizio del progetto, quasi tutti i requisiti sono rimasti invariati e alcuni sono stati leggermente modificati o annullati. Va notato che uno qualsiasi dei requisiti elencati di seguito è verificabile.

2.1 Requisiti di Business

Dal punto di vista degli obiettivi d'alto livello, il cliente si aspetta di:

1. Giocare ad un'applicazione simile ad Among us, videogioco *multiplayer* che può supportare da 4 fino a 10 giocatori;
2. Accedere ad una partita pubblica o privata già esistente oppure crearne una nuova;
3. Poter giocare una partita *multiplayer* contro altri giocatori "umani" secondo le regole base del gioco;

2.2 Requisiti Utente

Identifichiamo come utenti finali del sistema coloro che vogliano interagire con l'applicazione. Essi si aspettano dal sistema:

1. Creazione di una stanza specificando un codice ed il numero di giocatori;
2. Accesso ad una stanza di gioco già creata che può essere pubblica o privata;
3. Rappresentazione delle varie entità del gioco:
   - 3.1 *Mappa 2D*
   - 3.2 *Giocatori*
     - 3.2.1 *Crewmate*

3.2.2 *Fantasma Crewmate*
3.2.2 *Cadavere Crewmate*
3.2.2 *Impostor*

3.3 *Oggetti presenti sulla mappa*

3.3.1 *Collezionabili/task*
3.3.2 *Botole*
3.3.3 *Bottone di emergenza*

4. Visualizzazione di bottoni rappresentanti le abilità relative al proprio ruolo;

4.1 *Crewmate*

4.1.1 *Report*
4.1.2 *Emergency*

4.2 *Impostor*

4.2.1 *Kill*
4.2.2 *Report*
4.2.3 *Emergency*
4.2.4 *Sabotage*

5. Rappresentazione del movimento all'interno della mappa;
6. Visualizzazione della fase di votazione ed il risultato di essa;
7. Possibilità di abbandonare la partita;

2.3 Requisiti Funzionali

Dal punto di vista delle funzionalità, ci si aspetta che il sistema:

1. Permetta al Client di connettersi con il Server;
2. Gestisca più istanze di gioco contemporaneamente;
3. Raggiunto il numero di giocatori minimo, abiliti la possibilità di iniziare la partita;
4. In caso di disconnessione dell'utente, il giocatore verrà automaticamente eliminato dalla partita contrassegnandolo come deceduto;
5. In base al numero dei giocatori, selezioni casualmente da 1 a 3 *Impostor*, gli altri saranno considerati *Crewmate*;
6. Al termine della partita, dichiari il vincitore o i vincitori e renda possibile l'inizio di una nuova;
7. Visualizzi una mappa 2D con visuale fissa;

8. Gestisca un campo visivo prestabilito in base al tipo di personaggio tramite cui visualizzare gli altri giocatori nelle vicinanze;

9. Differenzi le abilità e gli scopi dei giocatori in base al proprio ruolo:

    9.1 *Crewmate*:

        9.1.1 Raccogliere oggetti collezionabili;
        9.1.2 Segnalare eventuali cadaveri dei *Crewmate*;
        9.1.3 Chiamare il bottone di emergenza per iniziare la fase di votazione;
        9.1.4 Segnalare il proprio voto;
        9.1.5 Morire a causa di un *Impostor* o tramite l'esito della fase di votazione;
        9.1.6 Diventare una fantasma dopo essere stato ucciso o eliminato;

    9.2 *Impostor*:

        9.2.1 Uccidere i crewmate senza farsi scoprire;
        9.2.2 Chiamare il bottone di emergenza per iniziare la fase di votazione;
        9.2.3 Segnalare eventuali cadaveri dei *Crewmate*;
        9.2.4 Segnalare il proprio voto;
        9.2.5 Effettuare dei sabotaggi;
        9.2.6 Usare le botole;
        9.2.7 Morire a seguito dell'esito della fase di votazione;
        9.2.8 Diventare una fantasma dopo essere stato eliminato;

2.4 Requisiti non Funzionali

2.4.1 Scalabilità

Il Server deve essere in grado di supportare un numero potenzialmente illimitato di giocatori suddivisi nelle varie partite.

2.4.2 Modularità

Il sistema è stato progettato in modo tale che, quando sarà necessario modificare il Server, il Client necessiterà di meno modifiche possibili e viceversa. In particolare, se il Server dovrà essere aggiornato, esso dovrà necessariamente essere spento e quindi riacceso. Gli utenti finali non dovranno eseguire alcuna operazione sui propri Client, o almeno si dovrà mantenere gli aggiornamenti il più piccoli possibile. Al contrario, in caso di aggiornamento del Client, non è necessario riavviare il Server o aggiornarlo. Tutto il software deve funzionare senza influenzare l'implementazione interna del modulo Core (regole e logica di gioco), questo se le modifiche non impatteranno sull'interfaccia.

2.4.3 Usabilità

Il sistema deve fornire agli utenti un'interfaccia chiara, semplice e ben organizzata in modo che essi possano sfruttare tutte le funzionalità realizzate.

2.4.4 Reattività

Data l'esecuzione di algoritmi di verifica, controllo e protocolli di comunicazione, il sistema deve essere in grado di consentire lo svolgimento delle partite senza ritardi e/o vincoli di tempo.

2.5 Requisiti di Implementazione

Le principali tecnologie utilizzate sono:

1. **Scala**: Il sistema deve essere prevalentemente sviluppato in Scala;
2. **Swing**: Il sistema deve disporre di un'interfaccia grafica che permetta di interagire con il gioco e con il Server;
3. **Akka**: Utilizzo di varie componenti utili per implementare un Server ed un Client che comunichino, secondo il paradigma di programmazione ad attori, per mezzo di messaggi asincroni;
4. **TDD**: si è tentato di applicare questa metodologia di sviluppo il più possibile all'interno del progetto. Si è fatto uso di *ScalaTest* ed *Akka TestKit*, utili per minimizzare la presenza di errori e di bug nel codice per facilitare l'espansione dell'applicazione;

Capitolo 3

Design Architetturale

3.1 Architettura generale

Una delle possibili rappresentazioni ad alto livello dell'architettura Client-Server del progetto AmongSus può essere rappresentato come in Figura 3.1:

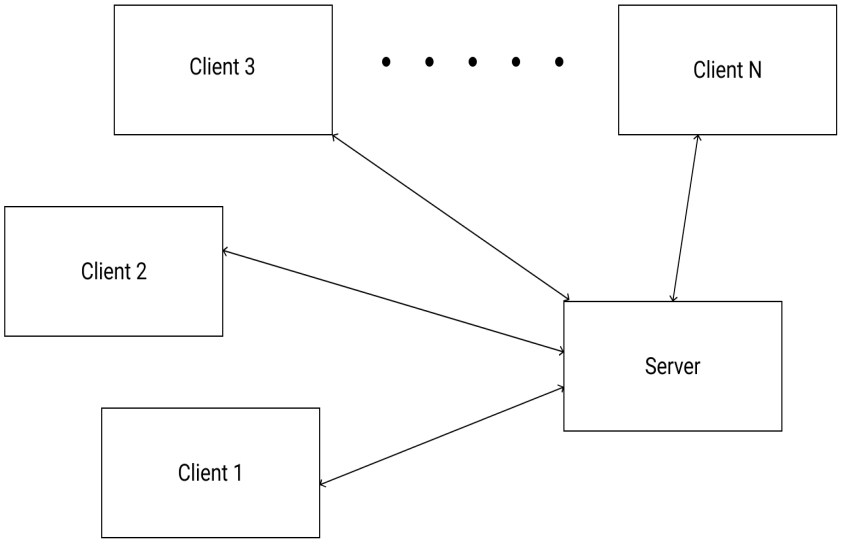

Figura 3.1: Rappresentazione architettura generale

Ogni giocatore viene rappresentato da un Client ed un unico Server si occupa della gestione delle connessioni tra giocatori e successivamente dello svolgimento della partita.

Questo particolare tipo di architettura è stata reputata migliore rispetto a quella Peer-To-Peer. In quest'ultima risulterebbe infatti più ostico garantire l'autenticità delle connessioni, la loro sincronizzazione e la distribuzione di eventuali informazioni. Nonostante il Peer-To-Peer avrebbe garantito una migliore scalabilità, si è ritenuto più opportuno scegliere l'architettura Client-Server.
Tramite quest'architettura si riesce comunque a soddisfare i requisiti non funzionali di scalabilità orizzontale. Oltre a questo si ha un vantaggio sia per quanto riguarda l'aspetto della sicurezza del sistema sia per l'aspetto della semplicità di sviluppo.

3.1.1 Core

Descrive unicamente, in maniera indipendente dagli altri moduli, la definizione delle entità base e delle regole che caratterizzano il gioco Among Us.
Espone delle API che possono essere utilizzate dai vari moduli dell'applicazione che dipendono da esso. Per tale ragione il Core può essere considerato come una vera e propria libreria esterna che non conserva alcuno stato relativo allo svolgimento del gioco.
In particolare, il Server lo utilizza per l'aggiornamento della partita in risposta alle mosse degli altri giocatori e il controllo della condizione di vittoria.
Il Client lo utilizza per la gestione interna del gioco, ovvero la validazione dei movimenti, le azioni dei giocatori nella schermata e per le interazioni con il Server relative agli aggiornamenti della schermata di gioco.

3.1.2 Server

Il Server è stato diviso in più componenti, i principali sono:

- **LobbyManagerActor**: Si occupa della gestione delle connessioni dei Client alle *lobby*. In particolare ha il compito di creare *lobby* pubbliche o private. Una volta raggiunto un numero sufficiente di partecipanti, genera un *GameManagerActor* che da quel momento in poi si occupa della gestione del gioco.
- **GameManagerActor**: Viene creato e richiamato per occuparsi della partita in corso. Quando il sistema è in fase di runtime, verranno creati molteplici attori di *GameManager*, uno per ogni *lobby*. Esso si occupa di ricevere le varie attività dei partecipanti e di aggiornare quelli presenti dei cambiamenti della partita come, ad esempio, *Kill* e *Emergency*.

3.1.3 Client

Per la realizzazione del Client è stato utilizzato il modello ad Attori cercando di rispettare il pattern MVC. La soluzione è stata organizzata per meglio isolare e suddividere i vari compiti:

- **Model**: Si occupa della gestione dei dati, della logica e mantiene lo stato del gioco.
  Comunica tramite il *ModelActor* con il *ControllerActor* notificando i propri cambiamenti interni e ricevendo gli aggiornamenti degli altri Client.
- **View**: Gestisce la user interface durante tutte le fasi del gioco, aggiornandosi a seguito di input dell'utente e/o cambiamenti del model.
  La comunicazione avviene grazie allo scambio di messaggi tra il proprio attore *UiActor* e l'attore del Controller *ControllerActor*.
- **Controller**: Il Controller ha il compito di avviare l'applicazione ed istanziare il *ControllerActor* che deve sucessivamente svolgere le seguenti attività:
  - Connessione con il server;
  - Gestione dei messaggi tra gli attori.
  - Gestione della fase di lobby;
  - Gestione della fase di gioco;
  - Gestione dei Timer per le azioni a tempo del gioco;
  - Gestione della fase di votazione;
  - Gestione della fine della partita.

3.1.4 Architettura ad Attori

Durante la fase di analisi e pianificazione dell'applicazione, si è notato subito il bisogno di uno scambio quasi continuo di informazioni/messaggi tra le varie entità della architettura.
Per questo motivo si è optato per l'utilizzo di un modello ad Attori (vedi Figura 3.2), sfruttando il Framework **Akka**.
Questo modello permette di modellare problemi altamente concorrenti attraverso un modello a scambio di messaggi asincrono. La modellazione ad attori, grazie alla proprietà di isolamento tra enti computazionali distinti, e grazie ad una solida organizzazione gerarchica delle responsabilità, pemette di ragionare in maniera semplificata su problemi nei quali la concorrenza, distribuzione e gestione dei failure sarebbero altrimenti molto difficili da trattare.

Un Attore è un'entità autonoma che possiede un proprio flusso di controllo ed è caratterizzato da:

- **Nome**: Permette di riconoscere univocamente l'attore all'interno del sistema;
- **Immutabilità dello Stato Interno**: Lo stato interno degli attori è immutabile ed il cambiamento viene espresso tramite cambio di Behaviour;
- **Behaviour**: Permette di utilizzare l'Attore come se fosse una macchina a stati finiti che reagisce agli input, ovvero i messaggi, ricevuti;

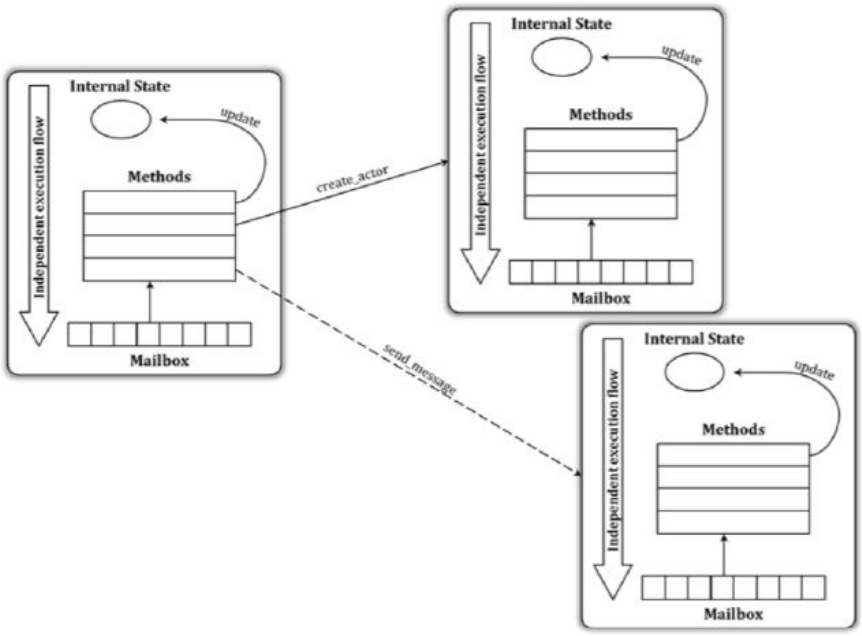

Figura 3.2: Rappresentazione concettuale degli Attori

L'Attore ha la possibilità di comunicare il risultato delle sue operazioni in tre differenti modi:

- Aggiornamento del proprio stato interno;
- Invio di uno o più messaggi ad un altro Attore;
- Creazione di un nuovo Attore;

Utilizzo Architettura ad Attori

Sfruttando il paradigma ad attori ci si è potuto concentrare sulle interazioni tra le entitá e studiare un'architettura che ottimizzasse le comunicazioni e lo scambio di informazioni. L'architettura derivante dall'applicazione di tale modello al nostro sistema ha portato a quella mostrata in Figura 3.3.

Comunicazione Client

- **ControllerActor**: Riceve le azioni compiute dall'utente tramite lo *UiActor*, comunica tutti i cambiamenti del gioco al *ModelActor*. Viceversa riceve notifiche riguardanti il cambiamento di stato del *ModelActor* e le inoltra allo *UiActor*;

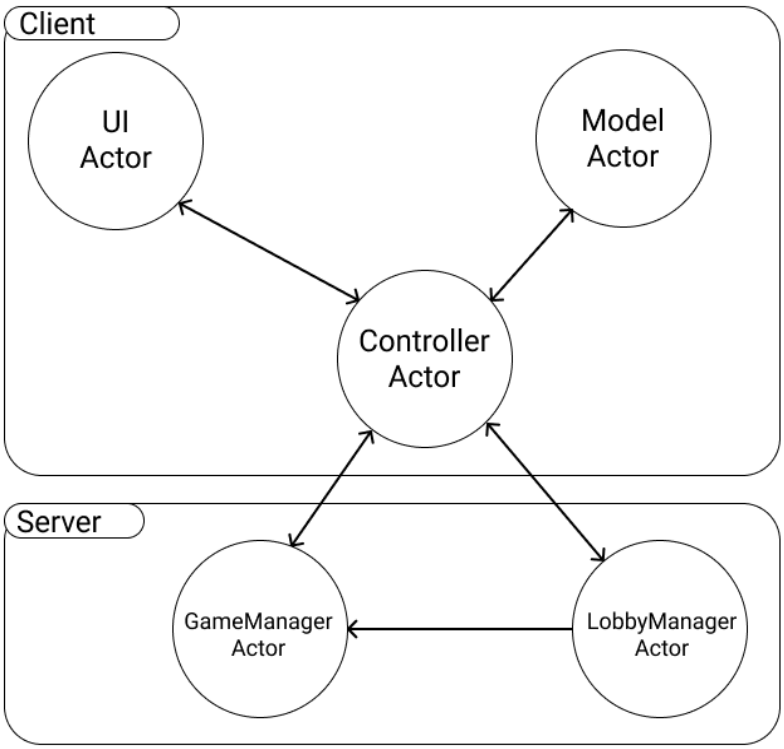

Figura 3.3: Architettura del Sistema Basata su Attori

- **UiActor**: Notifica il Controller delle azioni compiute dall'utente sull'interfaccia grafica, riceve notifiche dal *ControllerActor* e visualizza i cambiamenti nel Frame corrente;
- **ModelActor**: Riceve i messaggi che richiedono l'aggiornamento/modifica del proprio stato, effettua l'aggiornamento del proprio stato e lo notifica al *ControllerActor*;

Comunicazione Client-Server

- **ControllerActor**: Si occupa inoltre della comunicazione con il Server. In particolare notificando, ricevendo ed aggiornando lo stato del gioco al *GameManagerActor* o della *lobby* al *LobbyManagerActor*;
- **LobbyManagerActor**: Invia e riceve messaggi dal ControllerActor riguardo la gestione della *lobby*;

- **GameManagerActor**: Ha il compito di inviare e ricevere messaggi dal *ControllerActor* riguardo la gestione e l'aggiornamento dello stato del gioco tra i vari giocatori;

È stata fatta questa scelta per riuscire a garantire prestazioni ottimali durante l'esecuzione del gioco. Inoltre il sistema è stato modellato in maniera tale da prevedere modifiche future.

Capitolo 4

Design di dettaglio

In questo capitolo sono analizzate nel dettaglio le scelte effettuate dal team per la parte di Design del sistema e della sua realizzazione.

4.1 Organizzazione del Codice

La suddivisione del codice rispecchia il Design Architetturale generale descritto in precedenza ed è stato realizzato come in Figura 4.1:

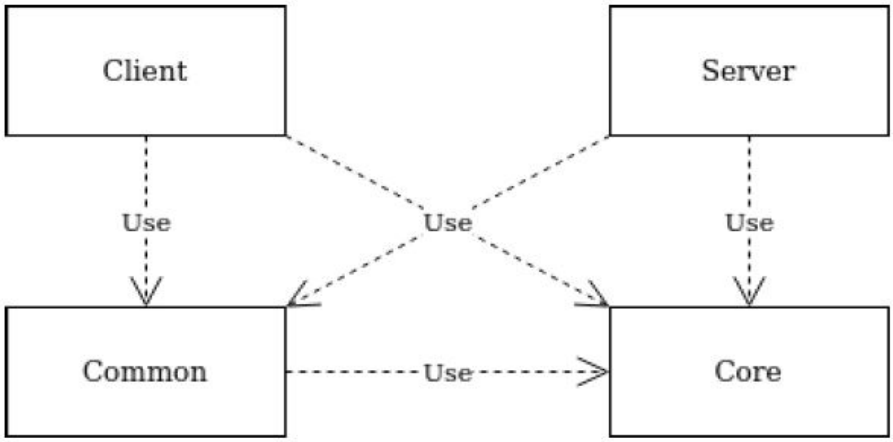

Figura 4.1: Suddivisione del Codice

Sono stati creati 4 moduli principali:

- **Client**: Il Client contiene tutta la parte relativa alle interfacce utente, la comunicazione con il Server e gli aggiornamenti dello stato della partita;

- **Core**: Questo modulo comprende tutte le entità che permettono la realizzazione del gioco e tutti i controlli che rendono possibile il proseguimento di una partita. Questo modulo è indipendente dal resto;
- **Common**: In questo modulo è presente il codice in comune per le parti di Client e Server. In particolare definisce i messaggi utilizzati per scambiare le informazioni;
- **Server**: Il Server contiene tutta la logica della gestione delle lobby e della varie partite in corso;

4.2 Core

Il Core si occupa della modellazione di tutte le entità del gioco Among Us, come i personaggi, le possibili abilità (*Kill* e *Sabotage*) e le funzionalità base del gioco (*Movement*, *Report* e *Emergency*). Serve, quindi, per permettere agli altri moduli del progetto di interagire con le entità. Queste funzioni implementano tutte le azioni che un giocatore potrà svolgere nel gioco reale.

4.2.1 Drawable

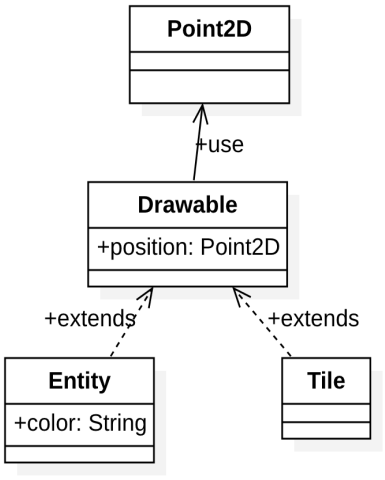

Figura 4.2: Class Diagram che rappresenta Drawable

Come si può vedere dalla Figura 4.2 le due entità principali del gioco sono *Tile* ed *Entity*. Entrambe estendono da *Drawable*, la quale rappresenta tutti gli oggetti disegnabili all'interno della mappa di gioco in una determinata posizione.

4.2.2 Tile

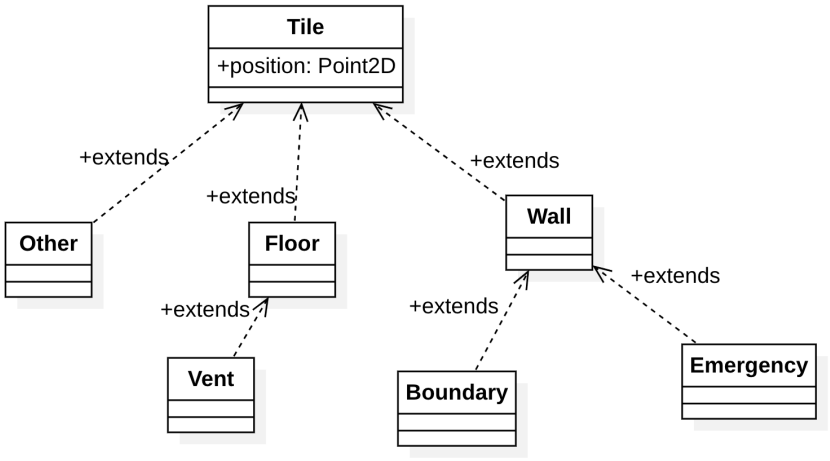

Figura 4.3: Class Diagram che rappresenta Tile

Tutti gli elementi statici che compongono la mappa estendono da *Tile* come mostrato nella Figura 4.3 e sono i seguenti:

- *Wall*: Rappresenta i muri della mappa di gioco, questi non potranno essere attraversati dai giocatori vivi ma solamente da quelli morti;
- *Floor*: Rappresenta il pavimento della mappa di gioco, sopra il quale tutti i giocatori potranno muoversi;
- *Vent*: Rappresenta una cella particolare del pavimento e permette agli Impostori di spostarsi rapidamente sulla mappa in quanto ogni Vent risulta collegata ad una sua "gemella";
- *Boundary*: Rappresenta il confine invalicabile della mappa oltre il quale nessun giocatore vivo o morto potrà andare;
- *Other*: Rappresenta gli spazi vuoti all'interno della mappa ai quali solo i fantasmi potranno accedere;
- *Emergency*: Rappresenta un particolare tipo di muro con il quale tutti i giocatori vivi potranno interagire una volta per partita;

4.2.3 Entity

Gli elementi dinamici che compongono la mappa estendono da *Entity* sono mostrati in Figura 4.4, ed a differenza della *Tile* possiedono anche un colore relativo

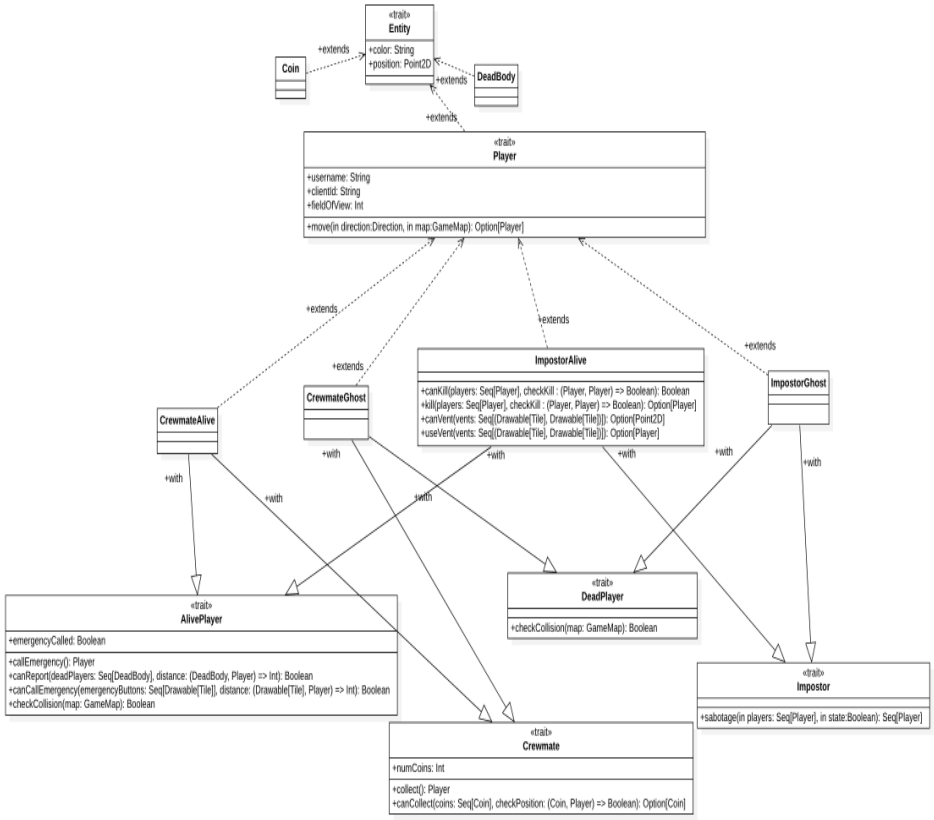

Figura 4.4: Class Diagram che rappresenta Entity

all'oggetto rappresentato. Tutti gli oggetti di questo tipo sono visualizzati sopra le *Tile* e sono i seguenti:

- *DeadBody*: Rappresenta il cadavere di un giocatore morto a causa dell'Impostore;
- *Coin*: Rappresenta tutti i collezionabili che possono essere raccolti dai Crewmate vivi e/o morti;
- *Player*: Rappresenta il giocatore base del gioco;

Player

Per quanto riguarda il Trait *Player* esso modella i tipi di giocatore presenti all'interno di una partita. Oltre al colore ed alla posizione ereditati da *Entity* aggiunge:

- *Username*: Campo che rappresenta il nome del giocatore che verrà renderizzato sopra al personaggio;
- *Client ID*: Campo univoco utilizzato per distinguere i vari giocatori;
- *Field Of View*: Campo di visuale variabile in base al tipo del giocatore;
- *Move*: Metodo che gestisce il movimento del giocatore all'interno della mappa;

I giocatori possono essere delle seguenti tipologie:

- *Crewmate Alive*: Rappresenta il giocatore buono che ha lo scopo di raccogliere tutte le *Coin* presenti nella mappa di gioco, può inoltre, in caso di ritrovamento di un cadavere, segnalarlo agli altri giocatori tramite la funzione *Report*. Ulteriormente, una volta per partita, può effettuare una chiamata di emergenza ed attivare la fase di votazione tramite l'*Emergency Button*;
- *Impostor Alive*: Rappresenta il giocatore cattivo che ha lo scopo di uccidere tutti i giocatori buoni della partita senza farsi scoprire e/o essere eliminato durante le varie fasi del gioco. Inoltre, può attivare un sabotaggio che gli permette di confondere i giocatori *Crewmate* ed utilizzare le *Vent* del gioco per spostarsi rapidamente da un punto all'altro della mappa;
- *Crewmate Ghost*: Rappresenta il giocatore buono ucciso durante una partita. Questo tipo di giocatore può solamente terminare la raccolta delle proprie Coin per portare la sua squadra alla vittoria;
- *Impostor Ghost*: Rappresenta il giocatore cattivo morto dopo una fase di votazione e la sua unica funzione è quella di aiutare i suoi compagni vivi utilizzando in modo strategico i sabotaggi;

Al fine di rendere modulare l'acquisizione di diversi comportamenti attraverso l'algoritmo di linearizzazione predisposto da scala, per la modellazione degli aspetti strutturali dei *Player*, si è sfruttato il meccanismo dei **Self-Type**.
Tale pattern di programmazione è un modo per dichiarare che un *trait* deve essere combinato tramite *mixin* in un altro *trait*, pur non estendendolo direttamente. Tramite questa tecnica è possibile fare *dependency injection* dichiarando esplicitamente nel *trait* le dipendenze di cui un componente necessita. Uno dei vantaggi della separazione del comportamento dalla struttura è quello della riusabilità dei concetti e quindi del codice. Mettendo assieme strutture e comportamenti è possibile combinarli e riutilizzarli per estensioni e implementazioni future.
Una volta definiti un set di proprietà ed i *self-type* corrispondenti, la definizione della componente strutturale dei *Player* si riduce ad un'attività di composizione di moduli elementari. Ciò porta ad una struttura composta da più livelli di componenti, alcuni dei quali condivisi da più *Player*. Questo approccio rende

riutilizzabili in futuro le astrazioni già definite permettendo di combinare i diversi comportamenti con flessibilità.
Focalizzando l'attenzione sull'implementazione mostrata nella Figura 4.4 la classe *CrewmateAlive* estende dal *trait Player* che ne definisce la struttura. Il comportamento è invece definito nei *trait Crewmate* e *AlivePlayer*. D'altra parte è possibile notare che anche *ImpostorGhost* mantiene la struttura derivata dal *Player*, ma il suo comportamento è descritto in *Impostor* e in *DeadPlayer*.
Per gestire il movimento dei diversi tipi di giocatori si è fatto uso del meccanismo delle **type class**. Il metodo *movePlayer* è stato reso generico e quindi utilizzabile da qualsiasi tipo di giocatore. Successivamente, in base al tipo di giocatore, viene fornita in maniera implicita una differente strategia di movimento. Inoltre, per ridurre il *boilerplate code* riguardante l'aggiornamento della posizione del giocatore, è stato utilizzato il pattern **Pimp my library** sulla classe *Point2D*.
Pimp My Library consente di estendere un tipo aggiungendovi metodi senza l'utilizzo dell'ereditarietà. Utilizzando questo Pattern si può aggiungere un metodo senza bisogno di estenderlo, attraverso conversioni implicite. Questa soluzione ha permesso di alzare il livello di astrazione del codice permettendo di ignorare i dettagli relativi alla costruzione della soluzione implementativa.

4.2.4 Prolog

Successivamente allo sviluppo delle funzionalità del core effettuato sfruttando il paradigma funzionale, tutto il team ha prodotto alcune di queste utilizzando la programmazione logica. L'integrazione è stata realizzata grazie alla libreria **TuProlog**.
In particolare è stata implementata la logica del movimento dei vari giocatori sia vivi che morti controllando tutte le possibili collisioni.
Oltre a questo sono state implementate alcune funzioni utili al calcolo della distanza fra due punti della mappa e fra un punto e lista di punti restituendo solo i punti della lista con distanza minore di un particolare input specificato.
Queste implementazioni possono tornare utili nel caso in cui è necessario controllare se l'impostore può effettuare una *kill*, oppure se un giocatore qualsiasi può effettuare l'operazione di report.
Tutte queste implementazioni non sono state richiamate nello sviluppo del gioco ma solamente testate nella classe *PrologTest*.
La teoria Prolog brevemente descritta è presente nel file *core/resources/theory.pl*.

4.3 Common

Il common è stato impiegato per raggruppare tutta la parte di codice in comune tra il modulo del Server e quello del Client. In particolare sono stati inseriti i vari messaggi che entrambi i moduli dovranno scambiarsi, qualche metodo per fare

logging e le costanti necessarie per il corretto funzionamento della connessione Client-Server.

4.4 Server

Questo modulo rende possibile il corretto svolgimento del gioco. Ogni sessione dei vari giocatori è possibile dividerla in due fasi:

- **Lobby**: Avviata l'applicazione, l'utente può connettersi al Server inviandogli i dati necessari per creare o entrare in una partita. Una volta connessi si viene inseriti in una lobby e si deve attendere finchè non si raggiunge un numero sufficiente di partecipanti per iniziare la partita. Al raggiungimento del numero minimo di giocatori (impostato in fase di avvio della partita), tutti i player sono rimossi e viene avviato il gioco;
- **Game**: Si occupa della gestione della partita. Nella fase di inizializzazione il Server comunica ai Client che è stata trovata una partita. Prima che il gioco cominci il server genera lo stato iniziale della partita e lo comunica ai giocatori. Successivamente si occupa di ritrasmettere i vari aggiornamenti di stato da parte di ciascun giocatore, andando a verificare se vi è presente una condizione di vittoria. Se una di queste condizioni è verificata, il Server notifica a tutti i giocatori il termine della partita, altrimenti invia un messaggio di aggiornamento;

4.4.1 Lobby

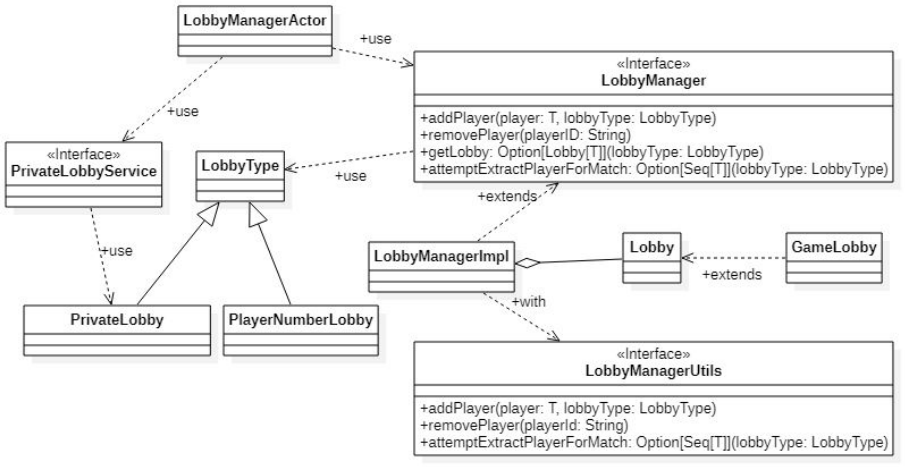

Figura 4.5: Class Diagram che rappresenta LobbyManagerActor

Per quanto riguarda la parte delle Lobby rappresentato nella Figura 4.5, è stato realizzato l'attore LobbyManagerActor che si occupa della gestione della

fase iniziale del gioco. Esso è stato creato estendendo il *Trait Actor* di Akka rendendo possibile la ricezione di messaggi provenienti dal modulo Client.
Buona parte della logica di questo componente è stata spostata fuori in altre strutture dati seguendo il più possibile il principio di *Separation of Concerns*.
La struttura base della Lobby è rappresentata da una lista di giocatori con le stesse impostazioni di gioco e con il medesimo numero di partecipanti. Per il *Trait Lobby* si è deciso di implementare il Pattern **Pimp My Library** implementando la classe *RichLobby* per una più attenta riorganizzazione della struttura.
Il componente *LobbyManager*, invece, è il componente che si occupa del mantenimento dei riferimenti a tutte le lobby create durante l'esecuzione del sistema. Esso espone i metodi per rimuovere, estrarre e creare giocatori da una determinata Lobby.
Per realizzare tutto ciò, *LobbyManager* mantiene una Lobby in corrispondenza di ogni *LobbyType* avente le informazioni sulle sue caratteristiche. Inoltre, si è deciso di implementare al suo interno il Pattern **Self-Type** per rendere modulari i comportamenti della Lobby.

Modellazione delle Private Lobby

Le *PrivateLobby* mantengono le medesime caratteristiche di una Lobby normale e, in aggiunta, possiedono un codice univoco per poter essere identificate. La Classe *PrivateLobbyService* si occupa del supporto alla creazione delle lobby private, generando un ID univoco ad ogni richiesta.

4.4.2 Game

GameManagerActor è l'attore del server che si occupa della gestione della partita, creato dall'attore *LobbyManagerActor* precedentemente descritto.
Questo Attore non mantiene lo stato globale della partita in quanto queste informazioni vengono mantenute all'interno di ciascun Client. Questa scelta è stata effettuata per ridurre il numero di messaggi scambiati tra Client-Server e ridurre il carico di lavoro del Server. Ciascun Client quindi prima di notificare il cambiamento del proprio stato esegue una serie di operazioni interne e, solo successivamente, notificherà il Server.
Lo svantaggio di questa architettura consiste nella possibilità da parte del Client di inviare uno stato fittizio al Server che non avrebbe alcun modo di verificarlo, andando a trasmetterlo a tutti gli altri giocatori.

4.4.3 Fault Tolerance

Per la realizzazione di tutte le componenti citate, si è cercato di gestire al meglio tutte le eventuali possibili situazioni di errore che si possono presentare, come ad esempio la disconnessione improvvisa di uno dei Client connessi. Per fare ciò si è fatto utilizzo delle funzionalità di supervisione e monitoraggio messe a disposizione da parte del Framework Akka.

Avendo a disposizione il riferimento dell'attore, è stato possibile mettersi in ascolto su un'eventuale evento di terminazione tramite messaggio *Terminated*. Queste funzionalità sono state sfruttate dall'attore Lobby per rimuovere gli utenti disconnessi dalle code.
L'attore responsabile delle partite, a fronte della terminazione di uno dei Client, notificherà tutti gli altri giocatori dell'avvenuta disconnessione, specificando l'ID del giocatore disconnesso.

4.5 Client

Per quanto riguarda il Client si è scelto di utilizzare un'**Architettura ad Attori** affiancata dal **Pattern MVC**.
Ogni componente ha quindi un proprio attore, il ModelActor è responsabile del Model, il *ControllerActor* del Controller ed infine lo *UiActor* per la parte di View.
Questa scelta ci ha consentito di ottenere all'interno del Client una buona separazione dei concetti tra le varie classi ed un'efficiente scambio di informazioni tramite i messaggi.
All'avvio dell'applicazione, tramite la classe *AppLauncher*, viene richiamato il *Controller*.

4.5.1 Controller

Il Controller ha il compito di istanziare e far partire il *ControllerActor*, rappresentato nella Figura 4.6, il quale deve gestire lo scambio di messaggi interno al Client e la comunicazione con il server.
Per gestire al meglio le diverse fasi previste dall'applicazione, si è optato per la creazione di diversi comportamenti che possono essere assunti dal *ControllerActor* durante l'esecuzione:

- **Lobby**: In questo comportamento l'attore ha come primo compito quello di stabilire una connessione con il Server. Per fare ciò deve andare a reperire l'indirizzo dell'attore del Server dal file di configurazione e generare il suo riferimento attraverso i metodi presenti in *LobbyInfo*. Successivamente si deve occupare della gestione dei messaggi inerenti alla fase di *match making* e della possibile terminazione dell'applicazione da parte dell'utente. Infine, istanzia il *ModelActor* una volta terminata la fase di *match making* modificando poi il suo comportamento in quello di *Game*;

- **Game**: Coordina tutto lo svolgimento del gioco aggiornando la View in seguito ad un cambiamento del Model, e viceversa. Essendo l'unico attore in grado di comunicare con il server riceve tramite messaggio tutti i cambi di stato degli altri giocatori notificandoli al Model. Nel caso in cui il proprio giocatore utilizza tramite la View il bottone destinato alla *Kill* o al *Sabotage* avvia un countdown avvalendosi della trait ActionTimer.

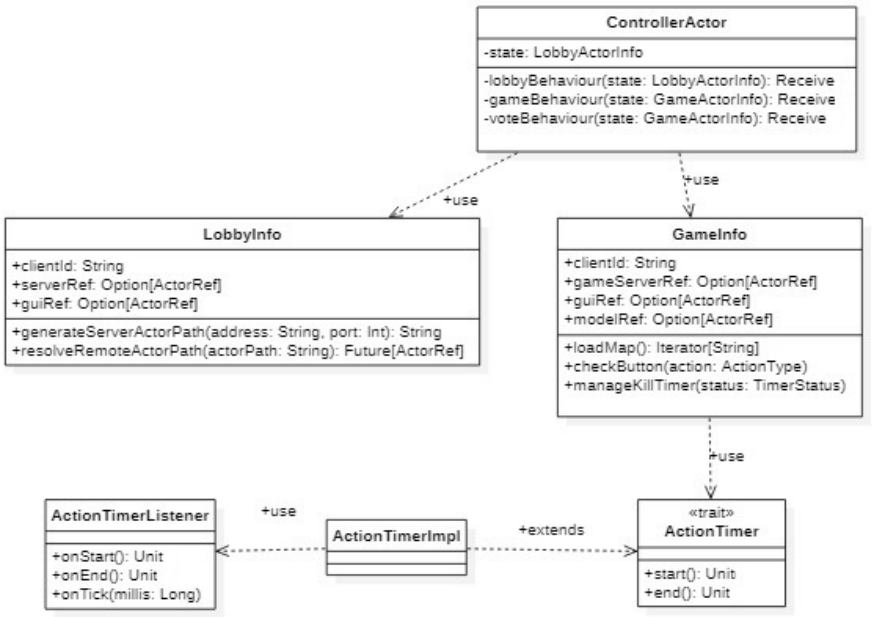

Figura 4.6: Class Diagram che rappresenta il ControllerActor

Diversamente se il giocatore preme *Report* o *Emergency* l'attore cambia il proprio comportamento passando a quello di *Vote*;

- **Vote**: Si occupa di quella fase di gioco in cui tutti i giocatori potranno votare un altro giocatore da eliminare. Ulteriormente a ciò, gestisce l'invio e la ricezione di eventuali messaggi nella Chat. Finita la fase di votazione ritorna nel comportamento del *Game*;

4.5.2 Model

Il Model nel Client è composto esclusivamente dal *ModelActor* rappresentato nella Figura 4.7, da *ModelActorInfo* utile per gestirne lo stato e da *ModelActorMessages* che contiene tutti i tipi di messaggio che è possibile ricevere.
Il *ModelActor* ha il compito di ricevere notifiche, da parte del *ControllerActor*, riguardanti azioni compiute dall'utente e/o di aggiornamento degli altri giocatori. In seguito alla ricezione di un messaggio di aggiornamento provvede ad aggiornare lo stato del Model ed a notificare a sua volta il *ControllerActor* del cambiamento di stato interno del Model.
Per fare ciò sono stati ideati due comportamenti:

- **Game**: Si occupa inizialmente della generazione della mappa e delle *Coin* destinate ai *Crewmate* facendo uso dei metodi forniti da *MapHelper*. In

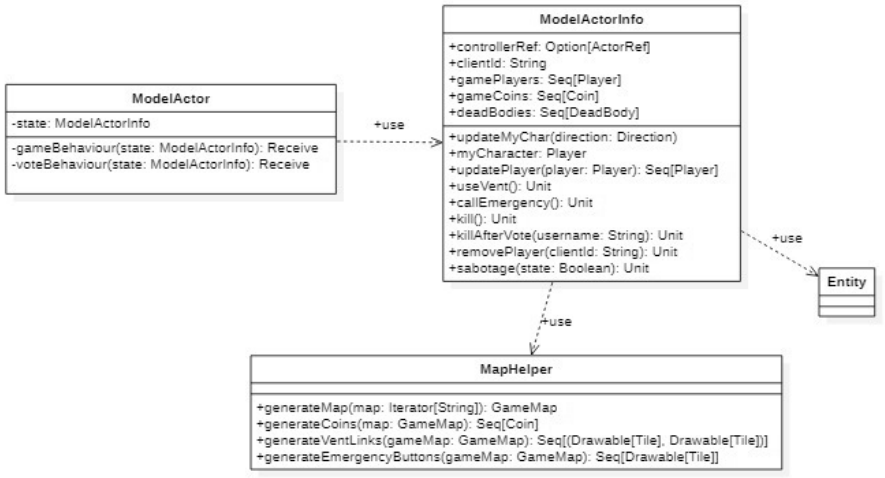

Figura 4.7: Class Diagram che rappresenta il ModelActor

particolare questo comportamento gestisce tutta la logica del gioco andando a verificare, utilizzando le funzioni del **Core**, se le azioni compiute da un giocatore sono valide o meno. Dopodichè va ad aggiornare lo stato del *ModelActorInfo*. In seguito verifica se il giocatore ha la possibilità di interagire con i vari elementi della mappa (*Vent*, *Emergency Button*, etc..) oppure con altri giocatori. Infine può cambiare il suo comportamento in Vote oppure, in caso di abbandono della partita da parte del giocatore, provvede a terminare se stesso;

- **Vote**: Questo comportamento si occupa dell'aggiornamento dei giocatori deceduti durante la fase di votazione, andando a notificare al Controller l'aggiornamento della lista dei giocatori deceduti e far tornare la partita nella fase di gioco;

4.5.3 View

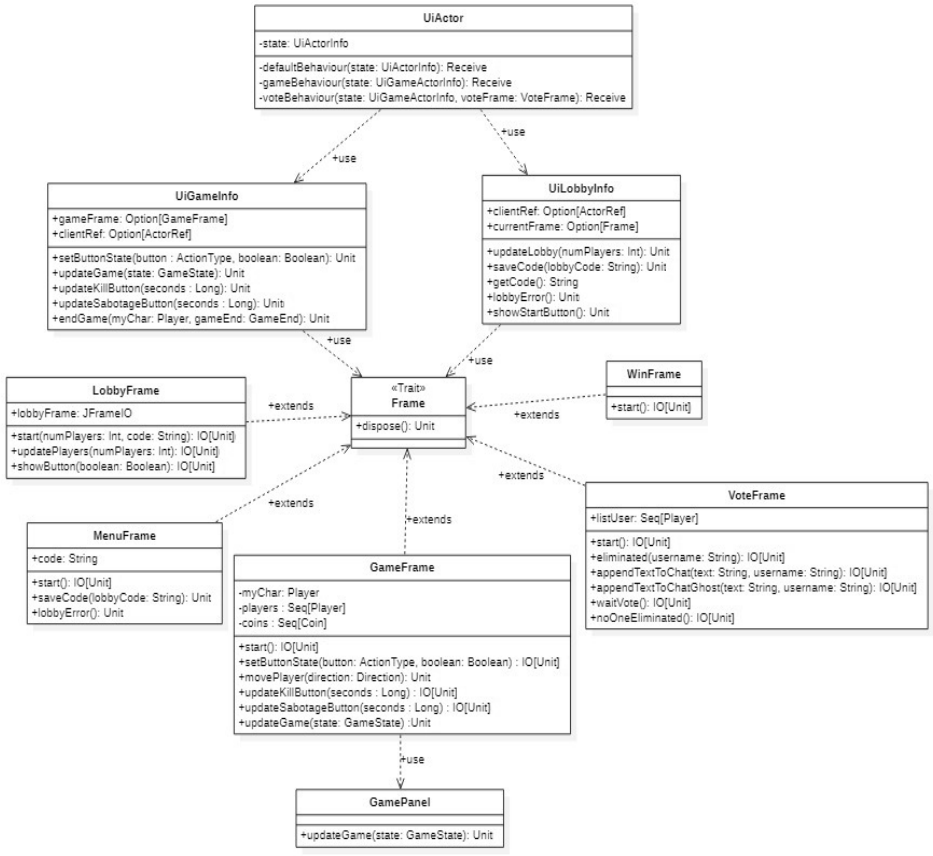

Figura 4.8: Class Diagram che rappresenta UiActor

In questo componente l'entità attiva è un attore chiamato UiActor, rappresentato in Figura 4.8, il quale comunica al *ControllerActor* le azioni dell'utente. Inoltre, aggiorna quando opportuno i vari Frame della GUI per mostrando i cambiamenti del Model.
L'*UiActor* può assumere uno dei seguenti comportamenti:

- **Lobby**: In questo comportamento l'*UiActor* è in grado di gestire tutte le operazioni di matchmaking. In dettaglio si occupa della gestione del *MenuFrame* e del *LobbyFrame*.
  Comunica con il *ControllerActor* ogni qualvolta l'utente voglia partecipare o creare una Lobby Pubblica o Privata. Visualizza e notifica all'utente l'entrata e l'uscita di giocatori nella Lobby. Inoltre, provvede alla generazione

del *GameFrame* una volta raggiunto il numero minimo di partecipanti e cambia il proprio comportamento in quello di *Game*;

- **Game**: È il comportamento che si occupa di notificare al *ControllerActor* tutte le azioni che il giocatore vuole effettuare. In particolare il giocatore può muovere il proprio personaggio tramite i comandi sulla tastiera oppure premere uno dei bottoni della GUI presenti nel *GameFrame*. Controlla il risultato del Timer delle azioni dell'impostore notificato dal *ControllerActor*. A seguito di un azione di tipo *Report* o *Emergency* cambia il proprio stato in Vote;

- **Vote**: Questo comportamento gestisce la GUI nella fase in cui l'utente deve esprimere il proprio giudizio selezionando un giocatore da eliminare. Inoltre visualizza gli eventuali messaggi inviati tramite la Chat all'interno del *VoteFrame*. In particolare, scambia alcuni messaggi con il *ControllerActor* relativi alla notifica del voto dei giocatori, la comunicazione del giocatore eliminato, l'invio e la ricezione dei messaggi relativi alla chat. Alla fine della fase di votazione cambia il proprio comportamento in *Game*;

Type class

Degno di nota è il modo in cui viene effettuato il rendering delle *Tile* e delle *Entity* per il quale è stato utilizzato il meccanismo delle *type class*.
Una *Type class* non è altro che un gruppo di tipi che soddifano un contratto tipicamente definito in un *trait*.
Questo permette di rendere una funzione polimorfica ad-hoc senza modificarne il suo codice. Infatti l'effettiva implementazione è definita in maniera implicita a seconda del tipo che richiama quest'ultima.
Questo pattern funzionale è visibile all'interno del package view.draw.

Wrapper Monadico di Swing

Per quanto riguarda lo sviluppo dei Frame presenti nella GUI, si è optato di utilizzare il framework **Cats.IO** per realizzare un *Wrapper* Monadico di **Swing**. Quest'ultimo permette l'accesso alle API più comuni dei principali componenti della libreria *Swing*. Ulteriormente, tramite questa implementazione è possibile sostituire il tradizionale metodo ad oggetti di chiamate ai metodi con un insieme di funzioni di tipo *IO Monad*. All'interno dei corpi di quest'ultime vi saranno tutte le operazioni necessarie alla realizzazione della GUI. Mentre tutti i side effect avverranno solamente al momento dell'esecuzione della monade. Inoltre, i risultati non saranno memorizzati, il che significa che il sovraccarico di memoria è minimo e anche che un singolo effetto può essere eseguito più volte in modo referenzialmente trasparente.
Un altro vantaggio derivante da questa scelta è la possibilità di riutilizzare il package contenente il wrapping monadico in altre applicazioni o di espanderlo con facilità in futuro in quanto risulta completamente indipendente dal resto del gioco.

4.6 Interazione Client-Server

Comunicazione fase Lobby

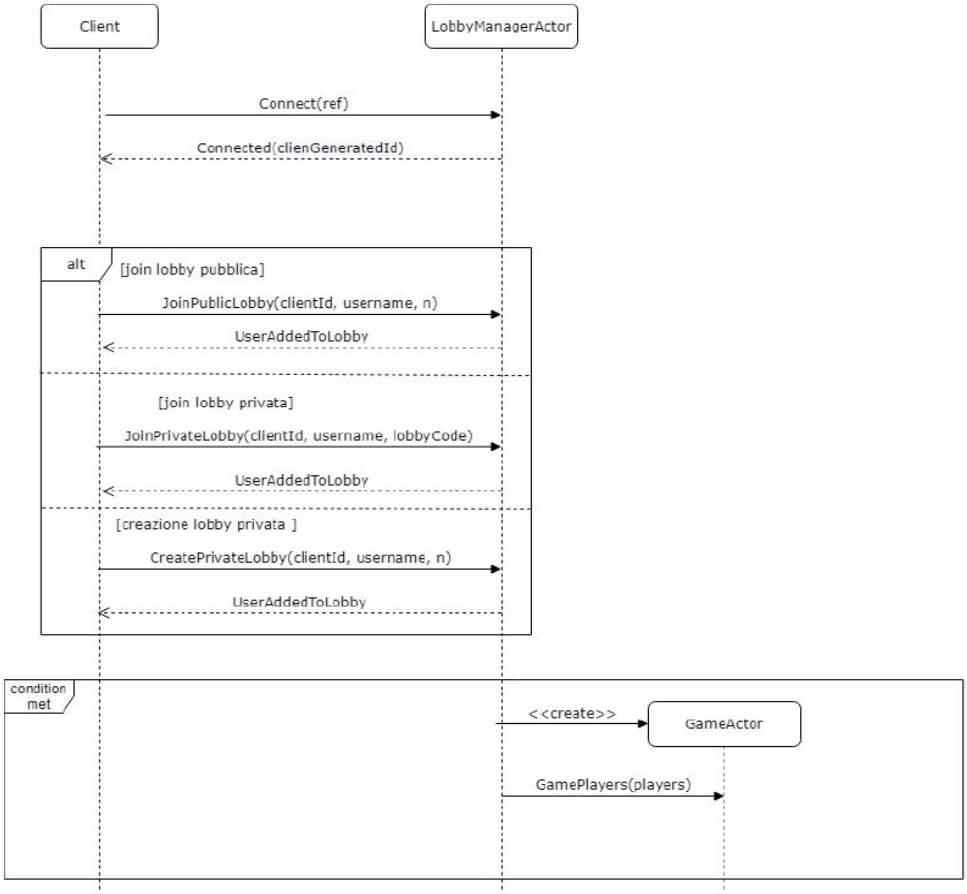

Figura 4.9: Sequence Diagram che rappresenta l'interazione con il Client

Per connettere un Client ad una Lobby e poter accedere e/o partecipare ad una partita, verranno eseguiti, come in Figura 4.9, i seguenti step:

- Il Client si occupa dell'invio di un messaggio di connessione al Server, in particolare al *LobbyManagerActor*, passando il riferimento all'attore Client a cui si deve rispondere.
- *LobbyManagerActor* notifica al Client l'avvenuta connessione e poi genera un ID univoco;
- L'utente imposta le sue preferenze di gioco ed il Client richiede al Server di essere aggiunto alla Lobby (Pubblica o Privata);

- Il Server aggiunge il Client alla lobby rispondendo con una notifica di avvenuta *Join*. In caso di Lobby Privata, viene inserito nella risposta un codice di accesso alla Lobby;
- Il Server, dopo l'aggiunta del giocatore alla Lobby, controlla se sono state verificate tutte le condizioni di gioco e, successivamente, procede con la creazione del *GameActor* per la gestione del gioco;
- Infine notifica al *GameManagerActor* la lista di giocatori connessi alla Lobby;

Comunicazione fase di gioco

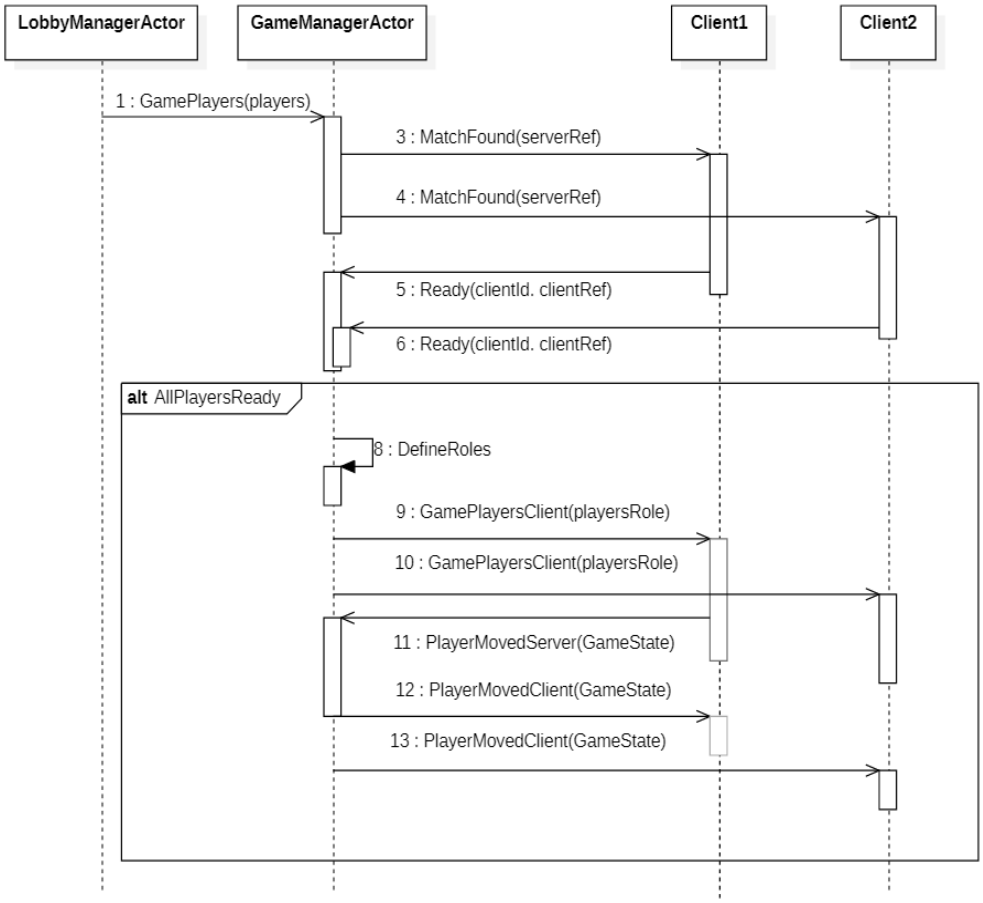

Figura 4.10: Sequence Diagram che rappresenta il comportamento del GameManagerActor

Una volta creato, l'attore responsabile della gestione della partita si occupa, come rappresentato in Figura 4.10, di:

- Rimenere in attesa del messaggio *GamePlayers* di inizializzazione contenente le informazioni dei giocatori;
- Notificare i player tramite un messaggio di *MatchFound* che la partita è stata trovata;
- Rimanere in attesa del messaggio di conferma *Ready* dei Client, avente l'ID del giocatore ed il riferimento dell'attore a cui inviare i messaggi successivi;
- Ricevere tutti i messaggi di avvenuta ricezione ed a quel punto inizializzare la partita;
- Generare la lista iniziale dei giocatori, scegliendo in base alle impostazioni uno o più Impostori ed inviandola ai vari Client connessi;

Il Client si occupa dell'intercettazione delle azioni eseguite dall'utente durante l'esecuzione del gioco e di comunicarle al Server, ricevendo inoltre le informazioni sullo stato di avanzamento del gioco.
Dopo aver inizializzato il gioco, il Server rimane in ascolto di eventuali azioni dei giocatori, che possono essere di due tipi:

- Messaggi di aggiornamento dello stato di un giocatore: in questo caso il server va a verificare se vi sono le condizioni di vittoria e notifica la fine della partita, altrimenti ritrasmette l'aggiornamento dello stato a tutti i giocatori;
- Messaggio che indica il passaggio alla fase di votazione: in questo caso il Server cambia il suo Comportamento passando dalla fase di gioco alla fase di votazione. In questa fase il Server ha la funzione di raccogliere i voti dai giocatori e di ritrasmettere i vari messaggi inviati tramite la Chat. Una volta raccolti i voti di tutti i giocatori viene selezionato l'eventuale giocatore candidato all'eliminazione se necessario. Viene successivamente controllata se vi è una condizione di vittoria, con conseguente notifica di termine partita, altrimenti il gioco torna al comportamento precedente notificando tutti i giocatori;

4.7 Interazione Attori Client

Nei diagrammi sottostanti sono rappresentati esempi di come avviene la comunicazione nel dettaglio tra gli attori presenti nel Client.
Il flusso di gioco del singolo giocatore viene gestito interamente in locale dal proprio Client.

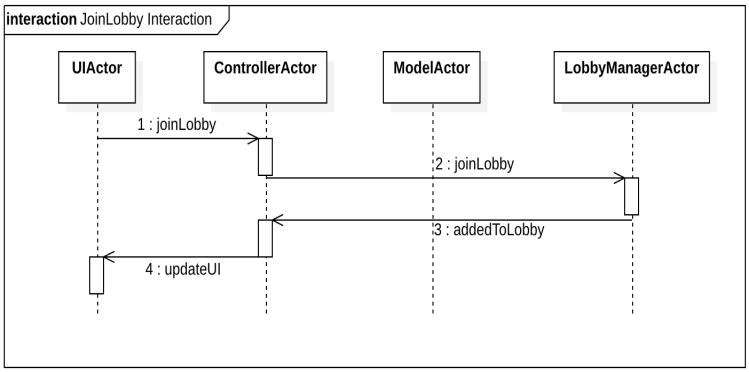

Figura 4.11: Sequence Diagram che rappresenta l'interazione nella fase *Join Lobby*

Nella Figura 4.11 è stato rappresentato come avviene l'interazione tra i vari Attori, sia Server che Client, nel momento in cui un player viene aggiunto alla Lobby. In particolare dallo UiActor viene inviato un messaggio di *joinLobby* al *ControllerActor* che notifica al *LobbyManagerActor* il *joinLobby* per aggiungere correttamente il nuovo *Player* alla lobby. Il *ControllerActor* riceve un messaggio di *addedToLobby* del completamento dell'operazione e notifica allo *UiActor* un *updatedUI* che permette l'aggiornamento della *View*.

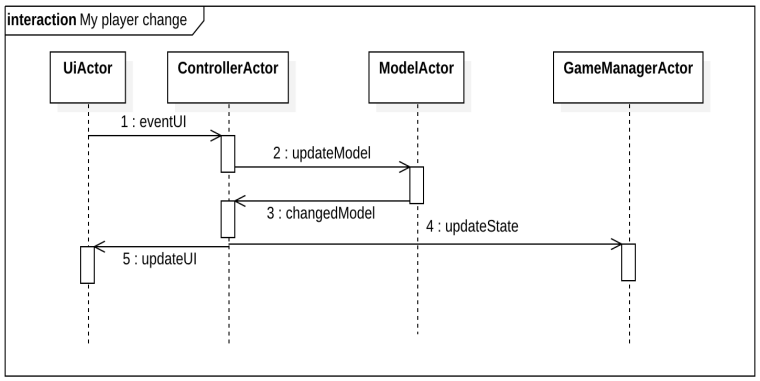

Figura 4.12: Sequence Diagram che rappresenta l'interazione a seguito di un azione da parte dell'utente

La Figura 4.12 rappresenta come il flusso di gioco sia gestito in locale. L'azione dell'utente proveniente dallo *UIActor* viene comunicata al *ControllerActor*. Il *ControllerActor*, dopo aver aggiornato il *ModelActor*, notifica il cambiamento al proprio UiActor. Oltre a questo il *ControllerActor* invia un messaggio anche al GameManagerActor per permettere di aggiornare anche tutti gli altri giocatori connessi.

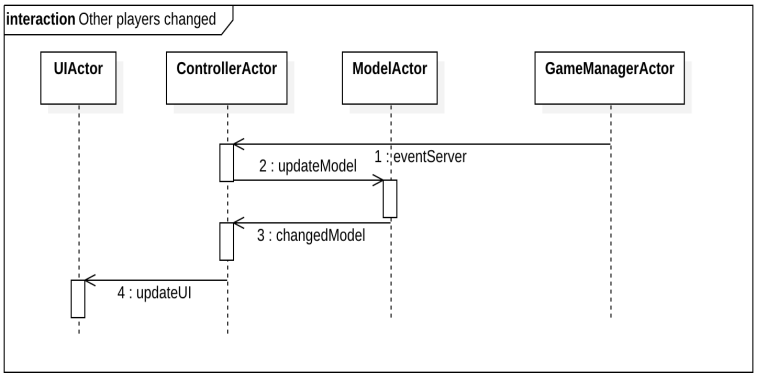

Figura 4.13: Sequence Diagram che rappresenta l'interazione a seguito di un messaggio dal server

Nella Figura 4.13 vi è rappresentato il flusso di informazioni generato a seguito di un cambiamento di un altro giocatore.
Il messaggio ricevuto dal *ControllerActor* indica che vi è stato un cambiamento di stato da parte di un qualsiasi altro giocatore. Il *ControllerActor* aggiorna lo stato in locale comunicando con il *ModelActor*.
Infine aggiorna lo *UiActor* dei cambiamenti effettuati.

Capitolo 5

Implementazione

Nel seguente capitolo verranno descritte nel dettaglio e motivate tutte le scelte implementative prese dai membri del Team.

5.1 Oleg

Nella fase iniziale del progetto mi sono occupato principalmente dell'architettura Client-Server e dell'organizzazione e suddivisione in moduli separati per cercare di ridurre le dipendenze all'interno del progetto. Ho lavorato anche sui file di configurazione di server e client per poter fare il *deploy* del nostro server sul cloud di AWS S3. In questo modo abbiamo potuto testare e provare se la nostra architettura si comportava come previsto.

Server

Per quanto riguarda la parte del server ho collaborato con Giovanni per organizzare la struttura e il comportamento degli attori *LobbyManagerActor* e *GameManagerActor*. In particolare ho gestito la fase di inizializzazione della partica, l'interazione durante essa e la gestione della condizione di vittoria. Ulteriormente, tramite le funzioni di libreria del framework *Akka* ho gestito la disconnessione da parte di un client.

Client

Ho collaborato con gli altri membri del team nell'organizzazione dell'applicazione secondo il pattern MVC e l'utilizzo degli Attori visto l'elevato scambio di messaggi previsto dall'applicazione. Successivamente, mi sono principalmente occupato della gestione della logica degli attori *ControllerActor* e *ModelActor*. Nel *ControllerActor* ho gestito la parte di comunicazione inerente alla fase di lobby e quella di gioco. Ulteriormente ho implementato la classe *ActionTimer* che è stata molto utile per gestire le azioni a durata del gioco. Mentre nel *ModelActor* ho gestito tutta la parte interazione nella fase di gioco. Questa parte

è stata più impegnativa rispetto alle altre in quanto prevedeva la gestione dello stato del gioco e della logica generale attraverso le API fornite dal modulo Core.

Insieme ad Elia abbiamo fatto un'implementazione puramente funzionale attraverso il *framework Cats*, di un intero package che permette di utilizzare componenti *Swing* in maniera funzionale.

Core

Assieme ad Elia abbiamo definito le varie entità che compongono il gioco e i loro comportamenti. In particolare abbiamo usato il pattern **self type** per la progettazione della parte strutturale dei vari tipi di giocatori. Inoltre il meccanismo delle **type class** per il metodo move dei diversi tipi di personaggio rendendo trasparente il differente modo di movimento di questi. Infine per aggiornare la posizione dei personaggi è stato applicato il pattern **pimp my library** aggiungendo al *Point2D* il metodo *movePoint* che permette data una direzione di aggiornare le coordinate del punto bidimensionale.

Infine ho creato il *trait MapHelper* con il relatovo **companion object** contenente le funzioni di *utility* per la generazione della mappa e degli oggetti che la compongono facendo utilizzo della **for comprehention**.

5.2 Elia

Ricoprendo il ruolo di *Product Owner* all'interno del progetto, mi sono occupato della coordinazione del lavoro dei miei colleghi. Inizialmente ho creato un *Product Backlog* dove ho inserito, per ogni task, una stima riferita alla difficoltà di sviluppo. Oltre a questo ho definito le priorità dei vari task, quest'ultime poi rivalutate con il resto del team di sviluppo. Grazie al *ProductBacklog* e all'utilizzo di *Trello* si è sempre tenuto sotto controllo lo stato dei lavori riuscendo quindi a rispettare tutte le scadenze degli sprint. Dopo aver definito insieme agli altri tutti i requisiti nella fase di analisi, ho iniziato a documentarmi su come implementare la view del gioco stesso. Per quanto riguarda lo sviluppo del progetto mi sono occupato principalmente dei moduli Client e Core.

Client

Ho definito con il resto del team tutta l'architettura ad attori, i relativi messaggi e l'applicazione di ciò seguendo il pattern MVC. Inizialmente mi sono dedicato allo sviluppo del *MenuFrame* e del *LobbyFrame*, grazie ai quali poi Oleg e Giovanni sono riusciti a testare l'integrazione delle funzionalità di matchmaking della view con il server. Per fare ciò ho gestito tutti i messaggi inviati e ricevuti dallo *UiActor*, mantenendone lo stato nella classi *UiGameInfo* e *UiLobbyInfo*. Ho partecipato ad una prima implementazione di alcune funzioni del *ModelActor*, ad esempio la generazione iniziale della mappa leggendo un file csv, che successivamente sono state spostate da Oleg nel *trait MapHelper*. Infine ho implementato singolarmente il *GameFrame* ed il *GamePanel* cioè la parte di view dedicata allo svolgimento del gioco. Successivamente nel *GamePanel* è stato

utilizzato il meccanismo delle **type class** per differenziare il metodo *draw* sia per le *Entity* sia per le *Tile*. Tutto ciò è stato implementato con Oleg nelle classi *DrawableEntity* e *DrawableTile*. La realizzazione di tutti i *Frame* descritti in precedenza ha fatto uso del package *view.swingio* che rende possibile l'utilizzo dei componenti Swing seguendo il paradigma funzionale. Quest'ultimo package è stato realizzato insieme ad Oleg.

Core

In questo modulo io ed Oleg abbiamo definito tutte le entità del gioco descrivendone i comportamenti. É stata effettuata una prima distinzione tra *Tile* ed *Entity*. Anche nel Core è stato utilizzato il meccanismo delle **type class** utile per distinguere il movimento in base al tipo di giocatore. Oltre a questo è stato utilizzato il pattern **self type** nella progettazione dei vari comportamenti dei *Player*. Infine è stato creata la classe *RichPoint2D* che aggiunge in maniera implicita il metodo *movePoint* alla classe *Point2D*.

5.3 Giovanni

Nella realizzazione delle componenti da me supervisionate del progetto, mi sono attenuto il più possibile allo stile di programmazione funzionale appreso durante il corso e focalizzando l'attenzione nel mantenere uno stato immutabile degli oggetto ove fosse più consono. Durante lo sviluppo del progetto, ho avuto l'occasione di lavorare sia al modulo del **Client** e, soprattutto, al modulo del **Server** nello sviluppo dell'architettura.

Client

Per lo sviluppo della parte relativa al Client, ho collaborato insieme ad Oleg ed Elia nella realizzazione. Mi sono occupato della creazione dei *Frame* e dei *Panel* relativi alla fase di gioco della Votazione, del processo relativo al Testing delle funzionalità principali ed alla realizzazione della *Scaladoc* per rendere più chiaro l'utilizzo dei metodi principali e per la generazione automatica della documentazione del codice sorgente scritto in linguaggio Scala. Nel **Model**, mi sono occupato del *Behaviour* della fase di Votazione, andando a creare l'apposito comportamento e gestendo tutti i vari *Case* relativi alle operazioni principali del gioco. Nella **View** ho provveduto alla creazione dei *Frame Vote* e *Win*. Grazie all'implementazione delle Monadi da parte di Elia e Oleg, ho potuto sfruttare appieno l'accesso alle API più utilizzate per la creazione delle GUI in *Swing* e la possibilità di effettuare chiamate ai metodi tramite un'insieme di funzioni di tipo *IO Monad*. Sfruttando la possibilità di riutilizzare i package contenenti il *wrapping* monadico, ho provveduto ad una veloce realizzazione del *Frame Win* per dichiarare semplicemente la vincita di un Team all'interno del Gioco e velocizzato il processo di creazione del *Frame Vote* per la creazione della Chat e della scelta sul giocatore da eliminare. Successivamente ho implementato il Comportamento della Votazione all'interno dello *UiActor* per una

corretta gestione di tutti i Case relativi alle notifiche ed operazioni di questa fase. Infine, insieme ad Oleg, all'interno del *ControllerActor* mi sono occupato della creazione e dell'implementazione del *Behaviour* della Votazione.

Core e Commons

Nei moduli Core e Commons ho aiutato Oleg nell'implementazione dei Test per verificare che tutte le funzionalità da lui implementate svolgessero correttamente il loro lavoro ed abbiamo provveduto alla creazione di classi per la gestione dei messaggi in comune tra Client e Server. Si è cercato di attenersi il più possibile ad una corretta metodologia di sviluppo, per permettere al Client ed al Server di comunicare efficacemente. Inoltre, ho aiutato Oleg ed Elia nella realizzazione del **Prolog** come Test delle funzionalità di base del modulo Core.

Server

Il Modulo del Server è la parte in cui mi son dedicato di più durante tutta la realizzazione del progetto. Insieme ad Oleg, mi sono occupato della realizzazione del *GameActor*. La mia attenzione si è focalizzata sul *Behaviour* della fase di Votazione che verrà gestita dal metodo *manageVote*. In questo metodo ho deciso di utilizzare le **For Comprehension** per sfruttare il più possibile il paradigma funzionale e rendere facilmente comprensibili le operazioni eseguite. Quando l'Attore riceverà un messaggio di *SendTextChatServer*, si occuperà del controllo relativo alla categoria del giocatore e reindirizzerà il messaggio filtrando solamente i giocatori di quella tipologia e notificando i rispettivi *ControllerActor*. Infine, dentro *GameActor*, si è fatto uso di una **Higher Order Function** accettando una Funzione come parametro del metodo. La parte che ha richiesto di più in termini di implementazione e realizzazione è relativa alle Lobby Pubbliche ed al *LobbyManagerActor*. In queste classi ho cercato di impiegare Pattern Avanzati per migliorarne l'implementazione. Nella classe *LobbyManager* ho implementato il Pattern **Self-Type** permettendo al *Trait LobbyManagerUtils*, tramite *Mixin*, di aggiungere metodi al *Trait LobbyManager* senza estenderlo direttamente. Questa funzionalità permette di rendere disponibili i metodi di *LobbyManagerUtils* al *LobbyManagerActor*. Ho deciso di utilizzare questo pattern perchè il comportamento del *Trait LobbyManager* è stato ampliato in corso d'opera con i metodi di *LobbymanagerUtils* e, per rendere più comprensibile questa aggiunta, ho pensato che *Self-Type* in questo caso potesse risultare una buona applicazione. Nel *Trait* Lobby ho deciso di implementare il Pattern Funzionale **Pimp My Library**. Permettendo al *Trait* Lobby di aggiungere la funzionalità fornita dal Metodo *extractPlayersForMatch* presente in *RichLobby* e che permette l'estrazione dei giocatori per dare inizio ad una partita. Infine mi sono occupato, insieme ad Oleg, della realizzazione di vari Test relativi alle funzionalità principali del Server per verificare che tutte le implementazioni svolgessero il loro compito correttamente.

Capitolo 6

Retrospettiva

6.1 Preparazione iniziale

Il Team ha effettuato un primo processo di sviluppo, suddiviso in uno Sprint iniziale per confrontarsi sul Design Architetturale da adottare e si è discusso sulle interazioni dei vari componenti del sistema. In questo Sprint si sono anche definiti i successivi sviluppi del progetto. Si è deciso che ogni Sprint, ad eccezione di quello inziale di avvio, avrà la durata media di 1,5/2 settimane massimo. Durante questa fase ci si è confrontati più volte tramite le piattaforma di incontro (Teams e Discord) causa pandemia e si è definita l'architettura ed i requisiti base del sistema.
Inoltre si sono individuate le componenti principali del gioco e le loro interazioni. Così facendo si ha avuto l'opportunità di confrontarsi e discutere sulla migliore organizzazione degli Sprint successivi.
Infine si è configurato l'ambiente di sviluppo base assegnando ad ogni componente del team un compito in base agli interessi dimostrati. Buona parte della documentazione è stata prodotta in questa fase, così da poter consolidare tutti i requisiti ed i componenti discussi nelle riunioni effettuate. Ci si è inoltre informati sull'utilizzo di *GitHub Actions* e *Trello*.

6.2 Sprint 1

In questo Sprint ci siamo posti come obiettivo lo sviluppo delle seguenti funzionalità:

Server:

- Gestione connessioni dei Client;
- Gestione delle Lobby pubbliche e private;
- Gestione inizio partita;

- Generazione *GameActor*;

Client:

- Connessione al Server;
- Implementazione della gestione delle Lobby pubbliche e private;
- *Wrapping* monadico di alcuni componenti di *swing*;
- Creazione della GUI del menù iniziale;
- Creazione della GUI della *lobby*;

Nonostante il pesante carico di lavoro di questo sprint siamo riusciti a concluderlo nei tempi previsti dedicando più ore lavorative rispetto a quelle normalmente previste per uno sprint.

6.3 Sprint 2

Tutti gli item dello sprint precedente sono stati sviluppati entro i tempi previsti. In particolare si è iniziata la realizzazione di un'interfaccia GUI per la gestione del gioco, la visualizzazione di una mappa e l'implementazione del movimento tra giocatori. Per fare ciò sarà necessario iniziare lo sviluppo del Model. In dettaglio sono state svolte le seguenti implementazioni:

Core:

- Creazione mappa 2D e dei suoi componenti;
- Creazione del giocatore;
- Gestione del movimento del giocatore.

Client:

- Movimento base del proprio giocatore all'interno della mappa di gioco;
- Gestione collisioni tra giocatori e ambiente;
- Comunicazione aggiornamento del movimento dei giocatori tra i vari client;
- Visualizzazione movimento di altri giocatori nella mappa di gioco.
- Gestione logica del gioco lato Client;

Server:

- Gestione logica del gioco lato Server;

6.4 Sprint 3

Il team è riuscito a sviluppare nei tempi previsti tutti gli item precedenti. In questo Sprint si è deciso di differenziare le abilitá in base al ruolo, la visuale dinamica e l'implementazione di tutte le abilità dei giocatori. Inoltre, essendo nei tempi, si è deciso di implementare la fase di votazione e la gestione del termine della partita.

Core:

- Differenziazione personaggi in base al loro ruolo;
- Gestione sabotaggi da parte dell'impostore;
- Gestione della *kill* da parte dell'impostore;
- Gestione della chiamata di *emergency* da parte di un giocatore;

Client:

- Visuale dinamica giocatore;
- Gestione delle azioni tra giocatori e giocatori;
- Gestione delle azioni tra giocatori e ambiente;
- Creazione GUI fase di votazione;
- Gestione della fase di votazione/gioco;
- Creazione GUI vittoria/sconfitta;
- Gestione fine partita;
- Dichiarazione team vincitore o perdente;
- Gestione Chat nella fase di votazione;

Server:

- Gestione della fase di votazione/gioco;
- Gestione condizione di vittoria;
- Gestione Chat nella fase di votazione;

6.5 Sprint 4

In questo Sprint, il team ha deciso di dedicarsi al refactor ed al miglioramento del codice del progetto. Sono stati effettuati i seguenti miglioramenti:

- Miglioramenti alla Scaladoc;
- Implementazione di test mancanti al fine di aumentare il coverage complessivo;
- Pulizia del boilerplate code;
- Aggiunta di pattern funzionali;
- Aggiunta di funzionalità extra non previste nei requisiti iniziali;

6.6 Sprint 5

Nell'ultimo Sprint, il team si è dedicato al completamento finale del progetto, apportando le seguenti modifiche:

- Completamento della relazione e controllo delle parti già realizzate;
- Utilizzo di Prolog per l'implementazione di alcune funzionalità del Core.
- Refactor e miglioramento della qualità del codice;
- Risoluzione ultimi problemi e bug;
- Caricamento del Server sul cloud di AWS S3;
- Release finale su GitHub del gioco e consegna;

Capitolo 7

Sviluppi futuri e Conclusioni

7.1 Sviluppi Futuri

Per quanto riguarda gli sviluppi futuri del progetto, il team ha pensato che si potrebbe ampliare l'applicazione inserendo alcune *feature* che potrebbero rendere più interattiva ed appassionante l'interazione dell'utente con gioco, ad esempio:

- Implementazione di una musichetta di sottofondo o dei suoni relativi alle varie azioni compibili dai giocatori nella mappa;
- Miglioramento degli effetti grafici dei frame e dei panel con animazioni più dettagliate o slider dedicati al progresso dello svolgimento dei vari task;
- Implementazione una *feature* extra che, all'uscita dal gioco di un giocare, sostituisca quel *player* con una IA che si occuperà dello svolgimento dei propri doveri per far raggiungere la vittoria al team;

7.2 Considerazioni finali

Il progetto è stata una bella esperienza formativa per il Team che si è trovato bene a lavorare con la Metodologia Agile appresa durante il corso. Avere obiettivi precisi da portare a termine in un tempo prefissato proporzionato alla capacità del Team ha contribuito ad alleviare lo stress dovuto all'avvicinarsi delle scadenze.
L'utilizzo di *Trello* come bacheca e di un *Product Backlog* ha contribuito a rendere molto più veloce ed intuitiva la comunicazione all'interno del Team in termini di presentazione dei problemi e dei lavori di ogni membro del Team.
Inoltre si è ampliata la conoscenza del linguaggio Scala e le sue potenzialità riguardanti lo sviluppo e la qualità software.

Concludendo, i membri del team hanno riscontrato un miglioramento delle rispettive capacitá di sviluppo in Scala, organizzative e di comunicazione nel lavoro in gruppo. Il team evidenzia che la suddivisione dei compiti è stata spesso bilanciata, la collaborazione e l'intesa tra i membri sono state un punto di forza. Tutti i membri, nonostante altri impegni universitari, si sono resi sempre disponibili nell'aiutare un membro e/o consigliare diversi modi di procedere.

Capitolo 8

Guida utente

8.1 Installazione ed Esecuzione

Per installare ed eseguire il Progetto, bisogna prima di tutto eseguire la **Build** tramite Shell, attraverso il seguente comando:

```
sbt clean pack
```

Successivamente si può procedere con l'esecuzione, eseguendo prima il comando che permette al **Server** di avviarsi:

- Windows:

  ```
  ./server/target/pack/bin/amongsus-server
  ```

- Linux/Mac

  ```
  .\server\target\pack\bin\amongsus-server
  ```

Ed infine il comando che permette di eseguire il gioco:

- Windows:

  ```
  ./client/target/pack/bin/app-launcher
  ```

- Linux/Mac

  ```
  .\client\target\pack\bin\app-launcher
  ```

Qui di seguito rappresentiamo alcune schermate del gioco:

Figura 8.1: Menù iniziale

Figura 8.2: Lobby partita privata

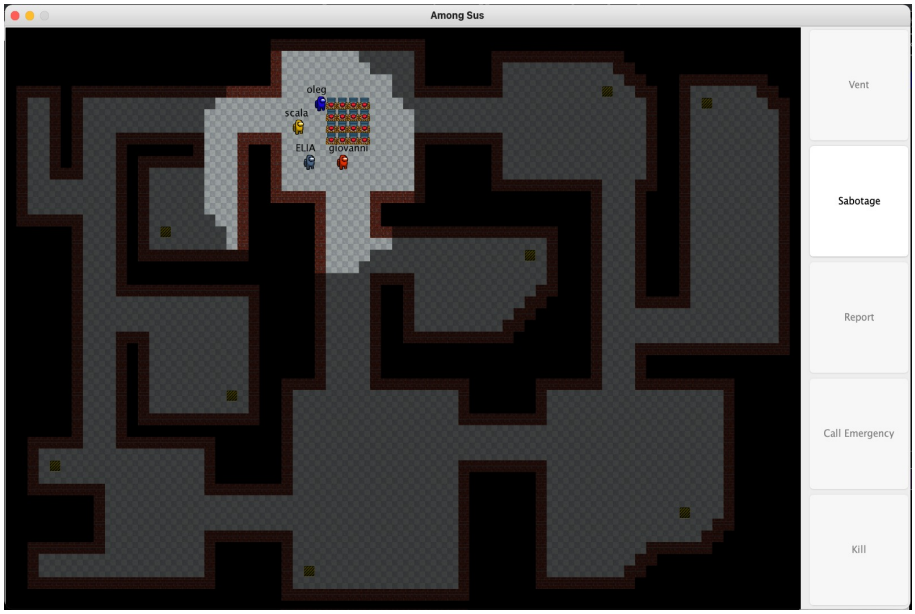

Figura 8.3: Partita con ruolo Impostore

Figura 8.4: Partita con ruolo Crewmate